

LARGE CODE BASE:

1. Read through:

- Documentation and examples, while using examples read API and browse through the source code, scanning the syntax, style of writing, deducing any patterns, etc;
- Read API thoroughly, imagine what the functionalities do; etc;
- Call the developers and ask questions about the system – design patterns, etc
- Determine if you are reading a framework or class of libraries. Patterns are still necessary to integrate with both, like one can use the strategy pattern on Django views to control the different model engines.

2. Coding principles:

- Is it functional or object oriented style;
- Verify that the principles have been followed: SOLID;
- ✕ *Check for inheritance, abstractions and dependencies – especially dependency injections; It is very critical to fully understand the problem being modelled so that we can easily define the classes/objects with their attributes and methods. Look for over-engineered solutions, solutions should be simple because requirements should be stated clearly. It is of no need to waste resources.*
- ✕ ***Read the codebase thoroughly, investigate it, follow it through manually, read documentations; then, use tools, Doxygen, to fill up the gaps.***
- ✕ *Find out how you will get it work, when given a problem get it to work first; in these you'll get to establish the critical data and methods required.*
- ✕ *Normally the concrete instances that varies are grouped together under an interface, abstract class or base class (inheritance), then a client class is created which combines the concrete objects in all sorts of different ways (normally via composition) and it defines its method interfaces' which calls the concrete objects' implementation method.*
- ✕ *Whether abstract classes are being used to manage subclasses, and a separate interface – mostly abstract classes are used to provide homogenous objects with a common interface;*
- ✕ *An interface must implement its own user intuitive methods, and through dependency injection calls the methods implemented by the subclasses that inherits the methods from the abstract class;*
- ✕ *Class methods can be used to break down an __init__ with modalities and create each instance separately for the code to modular and improve unit testing;*
- ✕ *Composition (initialise a class in an __init__ of another class) and Inheritance are the two standard techniques of reusing methods provided by the other classes;*
- ✕ *One can also pass a class into a method for dependency injection techniques and the class will be local to the object;*
- ✕ *Study the examples and tests. Inspect the classes or functions imported or locally defined that are involved in executing a particular example or test. That serves as an entry point to understand a particular functionality/component – you can use that to run gdb/pdb or to look up on Doxygen UML diagrams.*
- ✕ *Through tests and examples you get to learn how to use the classes or framework. It is also important to establish why the classes/framework was written, for hugging face the objective is to create models, for keras to create models, for django to run a web framework, etc; then you can begin your tests in those modules.*

- ✗ *UML Diagrams provide a structure, an arrangement of classes, from it we can then deduce the design approach for the problem.*
- ✗ *From all practical perspective software problems should be solved by iteration, different design, ad-hoc methods should be tried, tested and compared and select the one that closely satisfies the requirements.*
- ✗ ***Try understand each class libraries (subsystem) independently, focusing on the attributes, methods and local variables accepted. Then try to figure out the glue code between them to construct a framework, if it is a framework – how the classes use each other. Mostly this is found on examples and tests. Like how do we use the object provided? Glue code can be messy at first because it is meant to get the job done, and it mostly involves runtime objects – its getting the code to run! Normally this important glue code is hidden in utils, configs, decorators, etc – it is delegated some modules.***
- ✗ *Use `grep -r "import module_name" path/to/source_code` to search where a particular module has been used in the source code.*

- Check functions/methods:

*Understand why a function is there and what it accepts as inputs and returns as an output;
Check for data inputs and modality inputs;*

A function operates on the stack and returns values, if not, it modifies some aspect or interacts with the external world;

Functions should be written to minimise call stack to avoid cascading dependencies – use

__main__ attribute to control the function at one place, or modalities with default values;

We can also opt to write a separate piece of function that evaluates the modalities (separate functions), and inject the function (call the function directly from the main function) - it is still the same idea inline with dependency injection (we inject the functionalities into a client/interface function).

The modalities functionalities are not performed inside the function itself – interface and implementation have been decoupled. Break down the function into units even for testing purposes.

Functions may include loop and conditionals statements as part of their logic, data checks and modality checks.

3. Runtime environment:

- Try associate the runtime environment to understand how the software works and which functions are invoked.

There would be so many paths involved – think about it like a Gant Chart – some will be critical, others not;

Since examples and tests actually implement/test a particular feature that serves as an entry point – trace back the function calls to the one that actually returns the object;

There will be many hotspots in the code – find the functions that are called/call the most and try understand if they have a relationship;

4. Errors and constraints:

- From a library perspective, you want to ensure that you supply your users with useful error messages and to control how they use your objects.
You want to put fences around your objects and dictate how they should be used;

- From a user perspective you want to provide messages to those who will be running your scripts.

5. Interfaces

- Is the codebase designed as a library of classes or a framework? A library of classes contains distinct classes performing various functionalities mostly unrelated, and frameworks provide classes that already have glue code to coordinate between them and the application developer merely writes code to “complete” the framework – he provides missing pieces for the river to flow, like Django.

Tools:

1. Doxygen, ctags and cscope.
2. Sourcegraph and Github/1s/.
3. Sublime Text – LSP package with its language servers.

HOW GUNICORN WORKS

1. Fire up the terminal and type: *gunicorn <app> <host:port>*
2. Gunicorn has default settings for all the options, represented as classes in the file config.py. The list global variable: `KNOWN_SETTINGS` in the module `gunicorn.config` is populated via `gunicorn.config.SettingMeta` metaclass. Each class contains its default settings as class variables.

Metaclass code snippet: `def __new__(cls, name, bases, attrs):`

```

    super_new = super().__new__
    parents = [b for b in bases if isinstance(b, SettingMeta)]
    if not parents:
        return super_new(cls, name, bases, attrs)

    attrs["order"] = len(KNOWN_SETTINGS)
    attrs["validator"] = staticmethod(attrs["validator"])

    new_class = super_new(cls, name, bases, attrs)
    new_class.fmt_desc(attrs.get("desc", ""))
    KNOWN_SETTINGS.append(new_class)
    return new_class

```

Options’s class code snippet: `class Workers(Setting):`

```

    name = "workers"
    section = "Worker Processes"
    cli = ["-w", "--workers"]
    meta = "INT"
    validator = validate_pos_int
    type = int
    default = int(os.environ.get("WEB_CONCURRENCY", 1))
    desc = ""

```

The number of worker processes for handling requests.

A positive integer generally in the `2-4 x $(NUM_CORES)` range.
You'll want to vary this a bit to find the best for your particular

application's work load.

By default, the value of the ``WEB_CONCURRENCY`` environment variable, which is set by some Platform-as-a-Service providers such as Heroku. If it is not defined, the default is ``1``.

.....

the above class responsible for the option which sets the number of workers in the command line, and it is set to: `default = int(os.environ.get("WEB_CONCURRENCY", 1))` as default (if the option isn't used)

3. The module `gunicorn.__main__.py` is executed and it import `run` function from `gunicorn.app.wsgiapp`.

The `run` function initialises the `gunicorn.app.wsgiapp.WSGIApplication("%(prog)s [OPTIONS] [APP_MODULE]")` object. This object inherits from `gunicorn.app.base.Application` which in turn inherits from `gunicorn.app.base.BaseApplication`, and the `BaseApplication` signature is:

```
def __init__(self, usage=None, prog=None):
    self.usage = usage
    self.cfg = None
    self.callable = None
    self.prog = prog
    self.logger = None
    self.do_load_config()
```

will be executed to initialise the `WSGIApplication` object.

During this object's initialisation the method `self.do_load_config` will be executed which in turn executes:

```
def load_default_config(self):
    # init configuration
    self.cfg = Config(self.usage, prog=self.prog),
```

this method initialises the `gunicorn.config.Config` object to handle various configurations and set it to `self.cfg` variable. This is `Config`'s class initialisation signature method:

```
def __init__(self, usage=None, prog=None):
    self.settings = make_settings()
    self.usage = usage
    self.prog = prog or os.path.basename(sys.argv[0])
    self.env_orig = os.environ.copy()
```

as we can see it executes `make_settings` function from `gunicorn.config` and the function is defined as:

```
def make_settings(ignore=None):
    settings = {}
    ignore = ignore or ()
    for s in KNOWN_SETTINGS:
        setting = s()
        if setting.name in ignore:
            continue
```

```

    settings[setting.name] = setting.copy()
return settings

```

This function responsible for initialising the KNOW_SETTINGS list global variable. After calling load_config_default, it then calls, load_config which is not implemented in the BaseApplication class.

load_config method from gunicorn.app.base.Application will be executed to carry out the initialisation further. After the initialisation of the WSGIApplication its run method will be called. The run() method will be called and it will execute the line: self.load() which will execute the load method from the BaseApplication which is also not implemented, but implemented in: WSGIApplication and its signature is: def load(self):

```

    if self.cfg.paste is not None:
        return self.load_pasteapp()

```

```

    else:

```

return self.load_wsgiapp(), this function loads the wsgiapp through the method self.wsgiapp with the signature: def load_wsgiapp(self)

return util.import_app(self.app_uri), using the app_uri the app is then imported. The run method from Application class will ultimately execute: super().run() from the BaseApplication class. Which will then initialise the Arbiter: def run(self):

```

    try:

```

```

        Arbiter(self).run()

```

```

    except RuntimeError as e:

```

```

        print("\nError: %s\n" % e, file=sys.stderr)

```

```

        sys.stderr.flush()

```

```

        sys.exit(1)

```

At this stage the gunicorn.arbiter.Arbiter class will be initialised its initialisation signature is: def __init__(self, app):

```

    os.environ["SERVER_SOFTWARE"] = SERVER_SOFTWARE

```

```

    print("Find out which app")

```

```

    print(app)

```

```

    self._num_workers = None

```

```

    self._last_logged_active_worker_count = None

```

```

    self.log = None

```

```

    self.setup(app)

```

```

    self.pidfile = None

```

```

    self.systemd = False

```

```

    self.worker_age = 0

```

```

    self.reexec_pid = 0

```

```

    self.master_pid = 0

```

```

    self.master_name = "Master"

```

```

    cwd = util.getcwd()

```

```

    args = sys.argv[:]

```

```

    args.insert(0, sys.executable)

```

```

    # init start context

```

```

    self.START_CTX = {

```

```

        "args": args,

```

```

        "cwd": cwd,

```

```

    0: sys.executable
}

```

It will execute `self.setup(app)` method passing through the application which is `WSGIApplication`. This line: `self.cfg = app.cfg` will define the attribute `self.cfg` and assign `app.cfg` which is the `Config` class in `gunicorn.config` and it had already been defined at the function: `def`

```
load_default_config(self):
```

```
    # init configuration
```

`self.cfg = Config(self.usage, prog=self.prog)` in `BaseApplication`. The method `setup` will proceed and define the attribute: `self.worker_class = self.cfg.worker_class` where `self.cfg.worker_class` is `@property` method from `Config` class: `@property`

```
def worker_class(self):
```

```
    uri = self.settings['worker_class'].get()
```

```
    # are we using a threaded worker?
```

```
    is_sync = uri.endswith('SyncWorker') or uri == 'sync'
```

```
    if is_sync and self.threads > 1:
```

```
        uri = "gunicorn.workers.gthread.ThreadWorker"
```

```
    worker_class = util.load_class(uri)
```

```
    print(worker_class)
```

```
    if hasattr(worker_class, "setup"):
```

```
        worker_class.setup()
```

```
    return worker_class
```

This method will be executed later. Setup will then proceed to load the application: if

```
self.cfg.preload_app:
```

```
    self.app.wsgi()
```

this is the application argument passed in the command line arguments as `django.wsgi:application` in case of running a django application.

5. We will then execute `Arbiter.run()`, which executes the `self.start` method which will execute this line: `self.LISTENERS = sock.create_sockets(self.cfg, self.log, fds)` responsible for creating listeners on various sockets file descriptors, then the method `self.manage_workers` which will then executes `self.spawn_worker` (where this line: `worker = self.worker_class(self.worker_age, self.pid, self.LISTENERS`

```
        self.app, self.timeout / 2.0,
```

```
        self.cfg, self.log) links us to the worker classes in: gunicorn.workers)
```

which then uses `self.worker_class` defined during initialisation to the `worker_class` property method to load the worker from `workers.sync.Sync` (default and can be changed, based on this util function:

```
worker_class = util.load_class(uri)) (this is defined in part by the uri passed to the worker_class method of Config)
```

and it will be initialised by `Worker.__init__` from `gunicorn.workers.base` with signature: `def`

```
__init__(self, age, ppid, sockets, app, timeout, cfg, log):
```

```
    """\
```

```
    This is called pre-fork so it shouldn't do anything to the
    current process. If there's a need to make process wide
    changes you'll want to do that in ``self.init_process``.
    """
```

```
    self.age = age
```

```
    self.pid = "[booting]"
```

```
    self.ppid = ppid
```

```
    self.sockets = sockets
```

```
    self.app = app
```

```

self.timeout = timeout
self.cfg = cfg
self.booted = False
self.aborted = False
self.reloader = None

```

```

self.nr = 0

```

```

if cfg.max_requests > 0:
    jitter = randint(0, cfg.max_requests_jitter)
    self.max_requests = cfg.max_requests + jitter
else:
    self.max_requests = sys.maxsize

```

```

self.alive = True
self.log = log
self.tmp = WorkerTmp(cfg)

```

spawn_worker will proceed to execute: try:

```

    util._setproctitle("worker [%s]" % self.proc_name)
    self.log.info("Booting worker with pid: %s", worker.pid)
    self.cfg.post_fork(self, worker)
    worker.init_process()
    sys.exit(0)

```

which then execute `gunicorn.base.Worker.init_process` based on the line: `worker.init_process()`, which will execute `self.load_wsgi`, which then execute `self.wsgi = self.app.wsgi()`, the `self.app=from arbiter.setup(app)` which loads the proper wsgi handler.

Finally it executes: `self.run()` [

```

    # Enter main run loop
    self.booted = True

```

`self.run()`] to start the loop where `run` is not implemented on the base worker but on concrete workers, hence `run` will be executed by the subclass `gunicorn.workers.sync.SyncWorker`.

The above `run` method will execute `self.run_for_one`, which in turn will execute `self.accept` to accept the requests.

The `self.accept` method will call: `self.handle(listener, client, addr)` method and `self.handle` method will execute the line: `parser = http.RequestParser` which is used to receive and parse the message.

The `RequestParser`

```

(class RequestParser(Parser):

```

```

    msg_class = Request)

```

inherits from `gunicorn.http.parser.Parser` and here for the first time we see link to the `gunicorn.http` API [This API requires further investigation to disassemble exactly what the Parser does].

7. It then runs `self.handle_request` which creates

```

resp, environ = wsgi.create(req, client, addr, listener.getsockname(), self.cfg)

```

and `wsgi.create` is a function in `http.wsgi` and it relies on:

`resp = Response(req, sock, cfg)` for creating a response, here for the first time observe the relationship between the `Response` class and the `Workers` class, and this follows up:

```

responder = self.wsgi(environ, resp.start_response)

```

using `self.wsgi` variable assigned to our wsgi app to commence with the response. The `Response` class has `self.start_response` method. The `gunicorn.http.wsgi.Response` class requires further investigation.

DIVISIONS

1. We have to understand the strategy used by Gunicorn to parse command line parameters, handle environment variables and configuration files – how did Gunicorn implement these steps.
2. How socket listening works with multiple threads and differing server architectures from standard ones, like gthread to third-party ones like gevent.
3. How it parses the request – Parser class, how does this class work.
4. And how it prepares responses – Response class, how does this class work.

HOW DJANGO WORKS BEHIND GUNICORN & DEVSERVER

- ✗ *We will exclude the database logic in the meantime: we will focus on: commands, servers, routing, views, forms, templates, middleware and request & response.*
- ✗ *Insert import pdb and pdb.set_trace() inside functions, not outside them.*

1. From Gunicorn django enters the: `django.core.handlers.wsgi.WSGIHandler` class to start preparing for the response. From here (still presume), it will go through middleware, resolve url matching, create a request object, and then create a response.
2. `django.middleware.security` retruns: return `HttpResponsePermanentRedirect`, which has a direct link to `HttpResponse`
3. Standard Library `wsgiref.handlers.SimpleHandler` which inherits from `BaseHandler` runs the `__call__` to `WSGIHandler` to start processing the Response, as Gunicorn runs it directly from the `SyncWorker`.
4. The `render()` executed (found in `Template` class backends.django.py) is in the `base.py` `Template` class. More likely the context object (form class) has been processed when instantiating the `for` class inside the view method, because the class will be executed and it will be an object inside view – the class exist in `forms.py` module, same principles with instantiating models from `models.py`.

Steps

- ➔ We execute the `manage.py` file in the Django's project directory using the command line:
`python manage.py runserver`
Explanation:
- ➔ The function **`manage.main()`** is executed which then executes the function **`django.manage.execute_from_command_line(sys.argv)`**
Explanation:
Output:
- ➔ The following code snippet is executed from `django.core.management.__init__`

```
def execute_from_command_line(argv=None):  
    """Run a ManagementUtility."""  
    utility = ManagementUtility(argv)  
    utility.execute()  
Explanation: This code snippet initialises the  
django.core.management.__init__.ManagementUtility class and then executes its method  
execute(). This execute() method runs some setups, which include creating the command
```


line arguments parser and parsing the flags and arguments. Ultimately for the case of the subcommand == runserver, the following is executed:

```
self.fetch_command(subcommand).run_from_argv(self.argv)
```

Output:

- ➔ The above will execute the method `fetch_command(self, subcommand)`
Explanation: which is responsible for the getting the `app_name` and the relevant subcommand which is (runserver in this case). It will then execute: `klass = load_command_class(app_name, subcommand)`, which is responsible for loading the relevant Command line class which handles runserver, the class is: `django.core.management.commands.runserver.Command(BaseCommand)`. This class handles the runserver subcommand, and it inherits from `BaseCommand`. Time to shed some explanation as of the design of Django's command line execution: We have `BaseCommand` class which all the other subcommands like, runserver, makemigrations, migrate, etc, inherits from. This design pattern encourages flexibility and extensibility because should we desire to add more subcommands, we just write concrete subclasses without any disturbance to the system.
- ➔ The above `fetch_command()` method was called by `execute()` and it returns the above initialised class for the runserver subcommand, which then calls: `run_from_argv(self.argv)`
Explanation: this function is not defined in the runserver instantiation of its Command class, class, but it is inherited from `django.core.management.base.BaseCommand` class.
`run_from_argv` will set up the environment (remember for processes the information that we mostly manipulate is populated by the fork system call when a new process is being created, this information includes among: environment variables, command line arguments, and process's data structure information), so during setup this is the information which is being manipulated.
- ➔ The above `run_from_argv` method will call the method: `self.execute(*args, **cmd_options)`
Explanation: `execute` is defined in the `BaseCommand` class and it will try to execute subcommand == runserver, by parsing some command line options first and even checking for migrations and call the method: `output = self.handle(*args, **options)`
Output:
- ➔ The above `handle` method is not defined in the `BaseCommand` class:

```
def handle(self, *args, **options):
    """
    The actual logic of the command. Subclasses must implement
    this method.
    """
    raise NotImplementedError(
        "subclasses of BaseCommand must provide a handle() method"
    )
```

which makes sense since the `BaseCommand` class is not a concrete class to execute subcommand commands, so by the object oriented programming MRO algorithm the `handle` method from runserver concrete class will be executed.
- ➔ The above `handle` function will proceed with some initialisations and ultimately calls `self.run()` method.
- ➔ The above `self.run()` method will execute `self.inner_run()`.
- ➔ The above `self.inner_run()` will be responsible for getting the handler and executing run.
Explanation: `handler = self.get_handler(*args, **options)`, this will execute `get_internal_wsgi_application()` from `django.core.servers.basehttp`, which will then execute:

```
def get_wsgi_application():
    """
```

The public interface to Django's WSGI support. Return a WSGI callable.

Avoids making `django.core.handlers.WSGIHandler` a public API, in case the internal WSGI implementation changes or moves in the future.

```
"""
django.setup(set_prefix=False)
return WSGIHandler()
```

To retrieve the `WSGIHandler` instantiated object from `django.core.handlers.wsgi`, its signature is: `def __init__(self, *args, **kwargs):`

```
    super().__init__(*args, **kwargs)
    self.load_middleware()
```

This class is responsible for loading middleware objects from the class `django.core.handlers.base.BaseHandler` via the method `self.load_middleware()`. Middleware is a vast section which we will return to for a thorough discussion. For now let's proceed with starting our django development server.

- ➔ Ultimately the initialised `WSGIHandler` object will be returned to `inner_run`, which will then proceed to execute the `run` function from `django.core.servers.basehttp`, and this will be responsible for starting our development server.
- ➔ The `run` function:

Explanation: its signature: `run(`

```
    addr,
    port,
    wsgi_handler,
    ipv6=False,
    threading=False,
    on_bind=None,
    server_cls=WSGIServer,
```

`):`

`wsgi_handler` argument is the `WSGIHandler` object obtained above, and our `server_cls` is the: `server_cls = WSGIServer`, these variables are class variables defined in the `Command(BaseCommand)` class for `runserver` including `port`, `addr`, etc.

This function will define: `httpd_cls = server_cls`, an `httpd_cls` local variable and initialises it to the `WSGIServer`. It will then define an `http` variable and initialises it to:

`httpd_cls(server_address, WSGIRequestHandler, ipv6=ipv6)`, since `httpd_cls` is actually a `WSGIServer` object waiting to be executed and its initialisation signature is: `def __init__(self, *args, ipv6=False, allow_reuse_address=True, **kwargs):`

```
    if ipv6:
        self.address_family = socket.AF_INET6
    self.allow_reuse_address = allow_reuse_address
    super().__init__(*args, **kwargs)
```

The `WSGIServer` class defined in `django.core.servers.basehttp` inherits from the standard library `simple_server.WSGIServer` and this class's signature will also be executed by the above `super()` function. The `WSGIServer` from `simple_server` inherits from `http.server.HTTPServer` from the standard library which then inherits from `socket_server.TCPServer` which then inherits from `socket_server.BaseServer`, and ultimately this is the signature that will be executed: `def __init__(self, server_address, RequestHandlerClass):`

```
    """Constructor. May be extended, do not override."""
    self.server_address = server_address
    self.RequestHandlerClass = RequestHandlerClass
    self.__is_shutdown = threading.Event()
    self.__shutdown_request = False
```

notably here is the `RequestHandlerClass` argument, in our case this class was bundled into the `*args` argument which is a tuple of args and unpacked again using `*args` passed to

`super().__init__(*args, **kwargs)` to positional arguments. In this case our `RequestHandlerClass` argument is: `WSGIRequestHandler` and this class defined in `django.core.servers.basehttp`. In this whole process our request handler will be initialised into: `self.RequestHandlerClass = RequestHandlerClass`.

- ➔ The `run` function will also call: `httpd.set_app(wsgi_handler)` to setup the app. This function will execute the method in:

```
def set_app(self,application):  
    self.application = application
```

this will setup our application to `wsgi_handler`, which is our `WSGIHandler` class from `django.core.handlers.wsgi`.

- ➔ Then `run` will execute: `httpd.serve_forever()`, this will execute `self.serve_forever()` from `socket_server.BaseServer`:

Explanation: `serve_forever` will use the `selectors` standard library module to register sockets of interest with their corresponding events, refer to `selectors` module for more in-depth explanation of how it operates.

which will then call: `self._handle_request_noblock()`

- ➔ `self._handle_request_noblock()` will call: `request, client_address = self.get_request()`, which will then call `accept` from `sockets` library to accept a connection, and this function is in the subclass `TCPServer`.

After connection accepted the method will then call: `self.process_request(request, client_address)`, which will call: `self.finish_request(request, client_address)`.

- ➔ `Self.finish_request` will initialise our `RequestHandler` class:

```
self.RequestHandlerClass(request, client_address, self)
```

Explanation: initialisation of the `RequestHandlerClass` which is the `WSGIRequestHandler` from `django.core.servers.basehttp`, this class inherits from `wsgiref.simple_server.WSGIRequestHandler(BaseHTTPRequestHandler)`, which then inherits from `BaseHTTPRequestHandler` from the same module, which then inherits from `socket_server.StreamRequestHandler(BaseRequestHandler)`, which then inherits from `BaseRequestHandler` from the same module, and this class has an initialisation signature:

```
def __init__(self, request, client_address, server):
```

```
    self.request = request  
    self.client_address = client_address  
    self.server = server  
    self.setup()  
    try:  
        self.handle()  
    finally:  
        self.finish()
```

ultimately this is what will be executed. The `self.handle` method will be executed from `django.core.servers.basehttp.WSGIRequestHandler` `handle` method.

- ➔ The above `handle` method will call: `self.handle_one_request()`

- ➔ And `handle_one_request()` will:

Explanation: this function will ultimately execute the following important lines: `handler = ServerHandler(`

```
    self.rfile, self.wfile, self.get_stderr(), self.get_environ()  
)  
    handler.request_handler = self # backpointer for logging & connection closing  
    handler.run(self.server.get_app())
```

`ServerHandler` will be initialised and this class is in `django.core.servers.basehttp` and it inherits from `wsgiref.simple_server.ServerHandler` which then inherits from `wsgiref.handlers.SimpleHandler` with the following initialisation signature: `def __init__(self,stdin,stdout,stderr,environ,`

```

        multithread=True, multiprocess=False
    ):
        self.stdin = stdin
        self.stdout = stdout
        self.stderr = stderr
        self.base_env = environ
        self.wsgi_multithread = multithread
        self.wsgi_multiprocess = multiprocess
and it inherits from BaseHandler from the same module.
This line: handler.run(self.server.get_app()) will get the application which is a
WSGIHandler application from django.core.handlers.wsgi module. And it will the execute
run method from BaseHandler: def run(self, application):
    """Invoke the application"""
    # Note to self: don't move the close()! Asynchronous servers shouldn't
    # call close() from finish_response(), so if you close() anywhere but
    # the double-error branch here, you'll break asynchronous servers by
    # prematurely closing. Async servers must return from 'run()' without
    # closing if there might still be output to iterate over.
    try:
        self.setup_environ()
        self.result = application(self.environ, self.start_response)
        self.finish_response()
    except (ConnectionAbortedError, BrokenPipeError, ConnectionResetError):
        # We expect the client to close the connection abruptly from time
        # to time.
        return
    except:
        try:
            self.handle_error()
        except:
            # If we get an error handling an error, just give up already!
            self.close()
            raise # ...and let the actual server figure it out.

```

This method has been reproduced here because it is the method that links the Django application and the server. It will setup the environment and calls: `self.result = application(self.environ, self.start_response)`, where `application` is our `WSGIHandler`. This call will execute `__call__(self, environ, start_response)` from `WSGIHandler` thereby commencing with our response. The argument of interest into this function is `start_response` this function is found in: `wsgiref.handlers.BaseHandler` class, and it is called by `__call__` above: at this line: `start_response(status, response_headers)`.

- ➔ This in a nutshell is the process that happens from the command: `python manage.py runserver` until we start servicing the request. In the next section we begin exploring servicing the request.
- ➔ Conclusions:
Design patterns: loose coupling is the theme of object oriented programming

Servicing the request

- ➔ In the previous section we have entered the `__call__` method of `WSGIHandler` class from `django.core.handlers.wsgi`. This class has a class variable: `request_class = WSGIRequest`,

which is then called on this line: `request = self.request_class(enviroin)`. This will begin the preparation of an `HttpRequest` object.

- ➔ The `WSGIRequest` class inherits from `HttpRequest` class in `django.http.request`. This class will be initialised (its initialisation is a process and we will get back to it), and the initialised request object will be stored in `request` local variable.
- ➔ Then a call is made: `response = self.get_response(request)` to `get_response` method with request object as an argument. Since `WSGIHandler` inherits from `django.core.handlers.base.BaseHandler` this method will be executed from this class.
- ➔ The `self.get_response` method will execute, `response = self._middleware_chain(request)` where `self._middleware_chain` is a list of middleware objects initialised by the method `self.load_middleware` of the `BaseHandler` class. This list contains various middleware classes which must be executed to inject the request object they include among:
`django.middleware.security`
- ➔ `django.middleware.security` (investigate more how the below method is called from the above)
- ➔ ultimately, the method: `self._get_response(self, request)` is called which is an internal used by `self.get_response` to get the job done. This method will call: `callback, callback_args, callback_kwargs = self.resolve_request(request)` to resolve the request and returns among all the callback view function defined in the application's `view.py` module.
- ➔ The method `self.resolve_request` calls `def get_resolver(urlconf=None)` from `django.urls`:

```
if urlconf is None:
    urlconf = settings.ROOT_URLCONF
return _get_cached_resolver(urlconf)
```

this function will call `_get_cached_resolver` from `django.urls.resolvers` which will return the initialised object of the class: `URLResolver(RegexPattern(r"^/"), urlconf)` of `django.urls.resolvers` (This class must be discussed separately)

ultimately, the method `self.resolve` from the above `URLResolver` object will be called and it will return the initialised object: `ResolverMatch`(

```
    sub_match.func,
    sub_match.args,
    sub_match.dict,
    sub_match.url_name,
    [self.app_name] + sub_match.app_names,
    [self.namespace] + sub_match.namespaces,
    self._join_route(current_route, sub_match.route),
    tried,
    captured_kwargs=sub_match.captured_kwargs,
    extra_kwargs={
        **self.default_kwargs,
        **sub_match.extra_kwargs,
    },
)
```

of `django.urls.resolvers` [This object warrants a discussion of its own], and the following line: `request.resolver_match = resolver_match` which will assign `resolver_match` class variable to the above `resolver_match` object. And `resolver_match` tuple will be returned to `_get_response` method of the `BaseHandler` class in `django.core.handlers.base`, the tuple is possible because of the `__getitem__` method in the `django.urls.ResolverMatch` class.

- ➔ Now that we have our view class function (or class which needs further investigation), which can run further `self._view_middleware` checks, in this list we have middleware like: `CsrfViewMiddleware`, `SessionMiddleware` and `SecurityMiddleware`, this middleware includes checking of the views itself and redirecting the corresponding templates for quick responses. [This middleware requires further investigation]

- ➔ This section is responsible in the method `self._get_response` is responsible for executing the view: if response is None:

```
wrapped_callback = self.make_view_atomic(callback)
# If it is an asynchronous view, run it in a subthread.
if iscoroutinefunction(wrapped_callback):
    wrapped_callback = async_to_sync(wrapped_callback)
try:
```

```
    response = wrapped_callback(request, *callback_args, **callback_kwargs)
```

the if response is None condition is important since it confirms that middleware passed, otherwise a built-in template will have been populated.

This calls: `make_view_atomic(self, view)` in the same class `BaseHandler`, then the line:

`response = wrapped_callback(request, *callback_args, **callback_kwargs)` for synchronous execution is called to execute the view. Now we are in the `views.py` file in the application directory.

- ➔ Conclusion:

Design patterns:

Routing needs thorough investigations, and most of the concepts are missing here, especially the link between `get_response` and its executor `_get_response`.

Execution of the View

- ➔ From this line: `response = wrapped_callback(request, *callback_args, **callback_kwargs)`, if the view function has a render shortcut method, render will be executed from `django.shortcuts.render`.

- ➔ This is render: `def render(`

```
    request, template_name, context=None, content_type=None, status=None, using=None
):
    """
```

```
    Return an HttpResponse whose content is filled with the result of calling
    django.template.loader.render_to_string() with the passed arguments.
    """
```

```
    content = loader.render_to_string(template_name, context, request, using=using)
    return HttpResponse(content, content_type, status)
```

it begins by calling `loader.render_to_string` from `django.template.loader`, which calls: `template = get_template(template_name, using=using)` or calls `set_template` depending on some checks, both which are also located from `django.template.loader`.

- ➔ `get_template` function calls `engines = _engine_list(using)`, `_engine_list` function which returns the engines being used: we have `DjangoTemplates` and `Jinja2` engines in files `django.templates.backends.django` and `django.templates.backends.jinja2` respectively, which both inherit from `BaseEngine` from `django.templates.backends.base`. Then for each engine we will call its `get_template` method and it will return:

`Template(self.engine.get_template(template_name), self)` in case of a django template. The `Template` class is specifically for the Django engine and it is located at: `django.templates.backends.django`

Attention must be sought in the initialisation of this class `Template`: the argument:

`self.engine.get_template(template_name)` is of particular interest. This argument is actually the `Template` base class defined in: `django.templates.base` and it is initialised to `self.template` = `template` class variable of the `Template` class in `django.templates.backends.django`, it holds it as an extended class (Composition principle of object oriented programming).

This class `Template` initialised to `self.template` is responsible for Parsing the template, in essence the module `django.templates.base` is 1122 lines long since it defines `Nodes` and `Parser` itself [This needs a thorough investigation of how the django template parser works]

- ➔ After the Template object from `django.templates.backends.django` has been initialised to the class engine `DjangoTemplates` `get_template` method. Which is ultimately returned to the function `get_template` from `django.template.loader`, and which returns to the function `render_to_string` in `django.template.loader`, then the function will execute: `template.render(context, request)` the render method from `Template`.

- ➔ Here is the render method: `def render(self, context=None, request=None):`

```
# import pdb
# pdb.set_trace()
context = make_context(
    context, request, autoescape=self.backend.engine.autoescape
)
try:
    return self.template.render(context)
except TemplateDoesNotExist as exc:
    reraise(exc, self.backend)
```

this method executes `make_context` from `django.template.context`, and this function is responsible for gathering and executing the context/dictionary passed into the template inside the view function/class [This requires further investigation as this is where forms come into play). Normally forms will be in `forms.py` module of the application as classes and initialised inside a view – it's a vast topic found in `django.forms`. However the function `make_context` interacts with the `Context` and `RequestContext` classes defined in: `django.template.context`.

- ➔ Then render will call: `return self.template.render(context)` this is called on the `Template` from `django.templates.base` to render the template and parse it [This is another world which requires further investigations]

- ➔ Conclusions:

Design patterns:

This section still requires attention